Informatik B

Vorlesung 4
Fehlerbehandlung

Rückblick

- Kontruktoren
- Vererbung
- Casts
- Abstrakte Klassen
- Interfaces
- Packages
- Sichtbarkeiten

Fehlerbehandlung

- Es gibt im wesentlichen zwei Fehlerklassen:
  - Programmierfehler (eigene oder fremde Implementierung)
  - Nicht absehbare kritische Situationen zur Laufzeit
- Wie kann man innerhalb einer Methode auf Fehler reagieren?
  - Schlechte Idee: Rückgabe eines Fehlercodes
    - Oftmals wird die Interpretation eines Fehlerwertes missachtet, da der Programmierer davon ausgeht ein Fehler in dieser Situation könne gar nicht auftreten
    - Der Programmfluss wird durch Abfragen der Funktionsergebnisse unangenehm unterbrochen
    - Der Rückgabewert hat zwei Bedeutungen
  - Abhilfe: Assertions und Exceptions

Assertions

- Deutsch: Behauptung/Zusicherung
- Vor- und Nachbedingungen innerhalb von Methoden aufstellen, die den korrekten Ablauf garantieren sollen
- Assertions werden beim Programmablauf von der JVM überwacht
- Ist eine Bedingung nicht erfüllt, wird ein Fehler ausgelöst
- Die ausgelösten Fehler sind vom Typ **`Error`**
- Nach einer Fehlerhaften Bedingung sollte das Programm nicht weiter ausgeführt werden

Assertions

- Assertions sollen die Programmierung testen
- Mit Assertions sollten keine von äußeren Einflüssen abhängigen Bedingungen zugesichert werden (z.B. Parameterwerte)
- Assertions werden oft am Ende einer Methode verwendet, um das berechnete Ergebnis zu prüfen
- Assertions sollen **nur** dann scheitern, wenn Fehler in der Programmierung vorliegen

Assertions

- Syntax: **`assert expression [:Message];`**
- **`expression`** ist ein boolscher Ausdruck, der **`true`** oder **`false`** liefern darf
- **`Message`** ist ein String zur optionalen Fehlermeldung
- Bei Programmabbruch mittels einer Assertion wird eine Fehlermeldung ausgegeben:

```
Exception in thread main java.lang.AssertionError:
    Message
    at classname.methodname(filename:linenumber)
```

- Assertions kosten Rechenzeit, daher können Sie aktiviert oder stillgelegt (Standard) werden

Assertions

- Zur Aktivierung aller Assertions wird beim Start der JVM der Parameter **`-ea`** (*enable assertions*) mitgegeben:

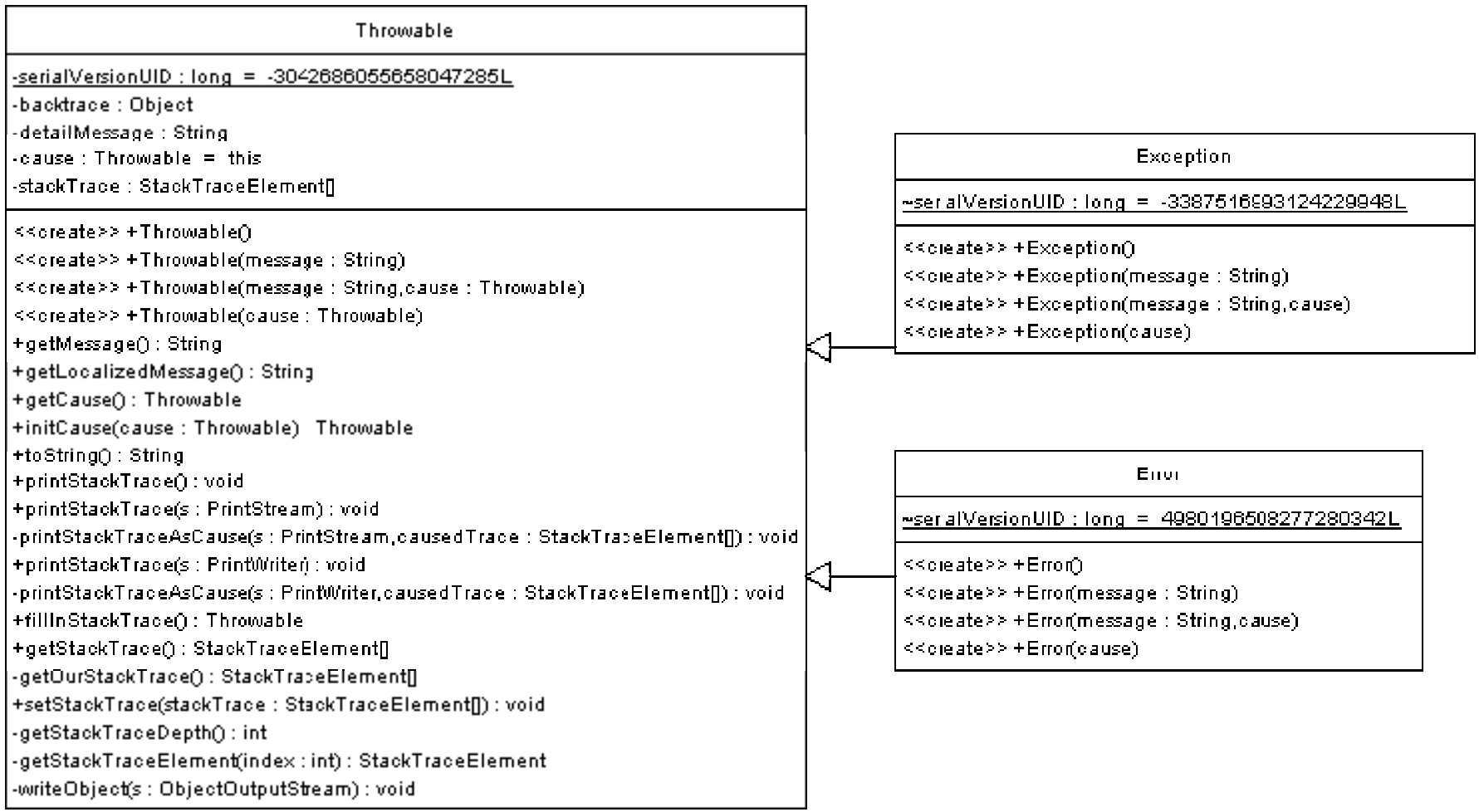
  ```
  java -ea Klassenname
  ```

- Assertions für einzelne Klassen können mit dem Parameter **`-ea:Klassenname`** aktiviert werden
- Assertions für ein Paket werden mit **`-ea:packagename…`** (Die drei Punkte kennzeichnen den Unterschied zu einer Klasse!) aktiviert
- Der Parameter **`-da`** (bzw. **`-da:classname`** oder **`-da:packagename…`**) deaktiviert die Ausführung von Assertions
- Mehrere Kommandozeilenparameter können kombiniert werden, um eine möglichst genaue Einstellung der Assertions zu erlauben
- Beispiel: **`assertion1`**

Exceptions (Klassenhierarchie)

- Eine Exception ist ein Objekt, dessen Typ direkt oder indirekt von der abstrakten Klasse **`java.lang.Throwable`** abgeleitet ist
- Von dort aus verzweigt sich die Hierarchie der Fehlerarten nach **`java.lang.Exception`** und **`java.lang.Error`**
- Bei einem Error handelt es sich um Fehler, die so schwerwiegend sind, dass sie zur Beendigung des Programms führen
- Der Programmierer sollte derartige Fehler nicht weiter beachten

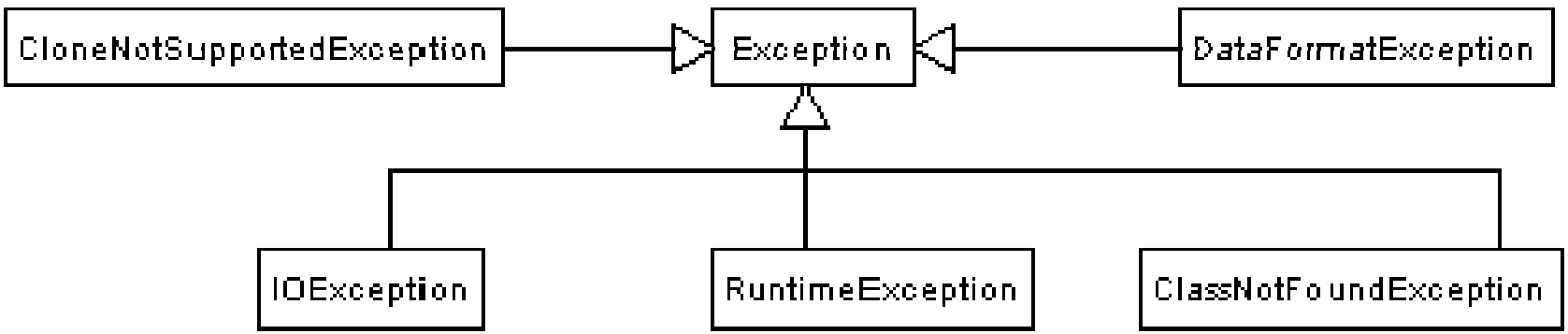
Exceptions (Klassenhierarchie)



Exceptions (Klassenhierarchie)

- Jede Benutzerausnahme wird von **`java.lang.Exception`** abgeleitet
- Exceptions sind Fehler oder Ausnahmesituationen, die vom Programmierer behandelt werden sollen
- Die Klasse **`Exception`** teilt sich dann nochmals in weitere Unterklassen beziehungsweise Unterhierarchien auf

Exceptions (Klassenhierarchie)



Exceptions

- Bei der Verwendung von Exceptions wird der Programmfluss nicht durch Abfrage des Rückgabestatus unterbrochen
- Ein besonders ausgezeichnetes Programmstück überwacht mögliche Fehler und ruft gegebenenfalls speziellen Programmcode zur Behandlung auf
- Den überwachten Block leitet das Schlüsselwort **`try`** ein und **`catch`** beendet es
- Hinter **`catch`** folgt der Programmblock, der beim Auftreten eines Fehlers ausgeführt wird, um den Fehler abzufangen (*to catch*) oder zu behandeln
- Es können mehrere **`catch`**-Blöcke angegeben werden, die unterschiedliche Fehler abfangen
- Es ist nach der Fehlerbehandlung nicht möglich, an der Stelle fortzufahren, an der der Fehler auftrat
- Beispiel: **`exception1`**

Exceptions

Ablauf eines Fehlers

- In einem Methodenaufruf wird ein **`Exception`**-Objekt erzeugt
- Die Abarbeitung der Programmzeilen wird sofort unterbrochen, und das Laufzeitsystem steuert selbstständig die erste **`catch`**-Klausel an
- Wenn die erste **`catch`**-Anweisung nicht zur Art des aufgetretenen Fehlers passt, werden der Reihe nach alle übrigen **`catch`**-Klauseln untersucht, und die erste übereinstimmende Klausel wird ausgewählt
- Die Spezielleren **`catch`**-Anweisungen müssen daher vor den allgemeineren stehen
- Es wird nur die erste passende **`catch`**-Klausel ausgeführt
- Beispiel: **`exception2`**
- Fehlerarten, die unterschiedlich behandelt werden müssen, verdienen immer getrennte **`catch`**-Klauseln
- Nur so ist eine sinnvolle Fehlerbehandlung möglich

Exceptions

`throws` im Methodenkopf angeben

- Will man in einer Methode nicht selber auf einen Fehler reagieren, kann man den Fehler weiter an den Aufrufer schicken
- Dazu wird im Methodenkopf eine **`throws`**-Klausel eingeführt
- Die Methode zeigt an, dass sie eine bestimmte Exception nicht selbst behandelt, sondern diese unter Umständen an die aufrufende Methode weitergibt
- Im Fehlerfall wird die Methode abgebrochen und eine Exception wird an den Aufrufer weitergegeben
- Der Fehler steigt so entlang der Kette von Methodenaufrufen wie eine Blase nach oben und kann irgendwann von einem **`catch`**-Block abgefangen werden, der sich darum kümmert
- Soll in einem Hauptprogramm keine Fehlerbehandlung durchgeführt werden, können alle Fehler an die Laufzeitumgebung weitergeleitet werden, die dann das Programm im Fehlerfall abbricht
- Beispiel: **`exception3`**

Exceptions

`throws` bei überschriebenen Methoden

- Überschriebene Methoden in einer Unterklasse dürfen nicht mehr Ausnahmen in der **`throws`**-Klausel deklarieren als schon bei der **`throws`**-Klausel der Oberklasse aufgeführt sind
- Das würde gegen das Substitutionsprinzip verstoßen
- Eine Methode der Unterklasse kann:
  - dieselben Ausnahmen wie die Oberkasse auslösen
  - Ausnahmen spezialisieren
  - weglassen

Exceptions

Abschlussbehandlung mit **`finally`**

- Nach einem (oder mehreren) **`catch`** kann optional ein **`finally`**-Block folgen
- Die Laufzeitumgebung führt die Anweisungen im **`finally`**-Block immer aus, egal, ob ein Fehler auftrat, oder die Anweisungen im **`try/catch`**-Block optimal durchliefen
- Das heißt, der Block wird auf jeden Fall ausgeführt, auch wenn im **`try/catch`**-Block ein **`return`**, **`break`** oder **`continue`** steht oder eine Anweisung eine neue Ausnahme auslöst
- Der Programmcode im **`finally`**-Block bekommt nicht mit, ob vorher eine Ausnahme auftrat
- Sinnvoll sind Anweisungen im **`finally`**-Block immer dann, wenn Operationen immer ausgeführt werden sollen
- Eine typische Anwendung ist die Freigabe von Ressourcen oder das Schließen von Dateien
- Beispiel: **`finally1`**

Exceptions

Das Duo **`return`** und **`finally`**

- Ein Phänomen in der Ausnahmebehandlung von Java ist eine **`return`**-Anweisung innerhalb eines **`finally`**-Blocks
- Ein **`return`** im **`finally`**-Block überschreibt den Rückgabewert eines **`returns`** im **`try/catch`**-Block
- Beispiel: **`finally2`**

Verschwinden von Exceptions

- Ist im **`finally`**-Block eine **`return`**-Anweisung vorhanden wird eine im **`catch`**-Block ausgelöste Exception nicht zum Aufrufer weitergeleitet
- Es wird lediglich der Rückgabewert zurückgeliefert
- Beispiel: **`finally3`**

RuntimeException

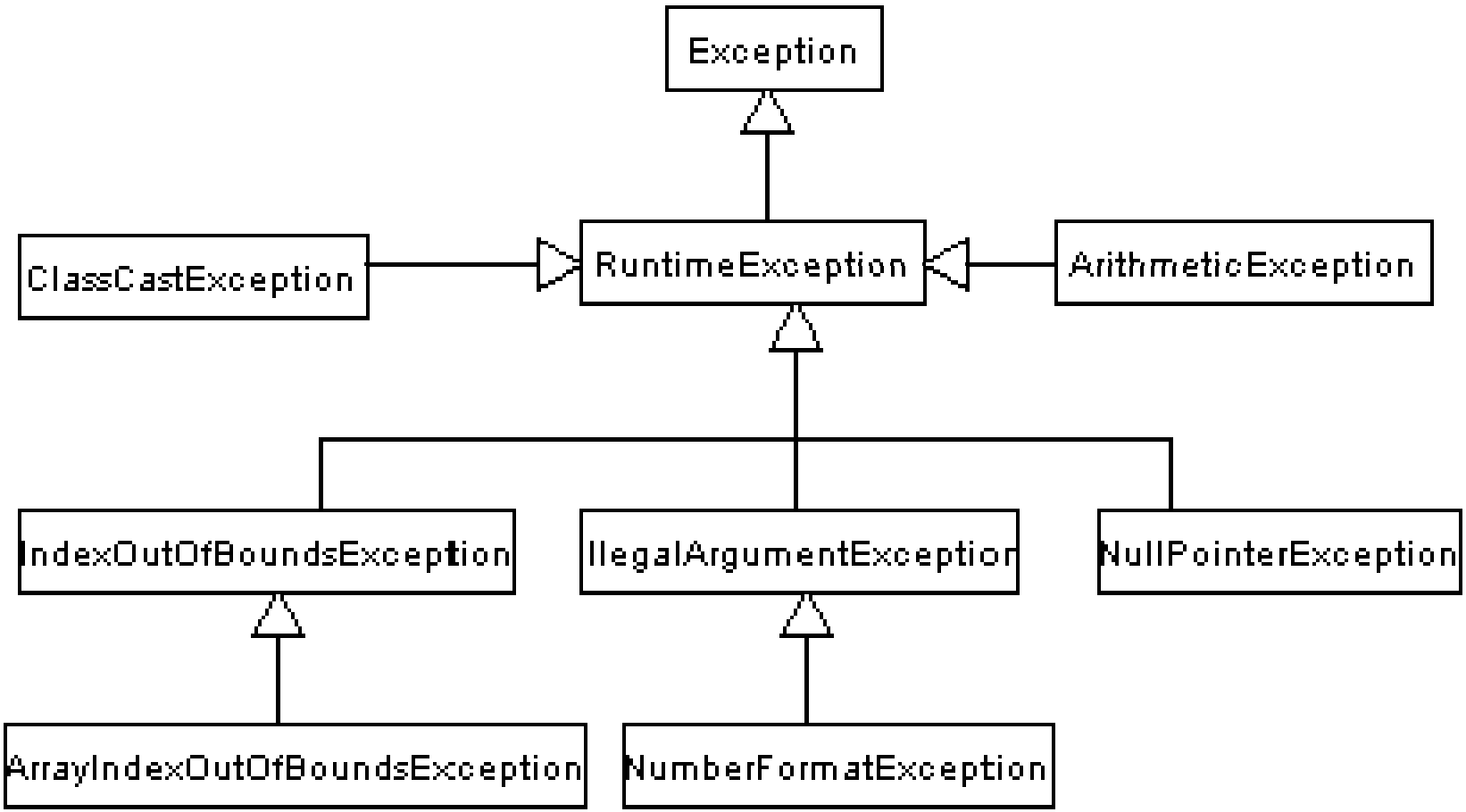
- Einige Fehlerarten können potenziell an vielen Programmstellen auftreten
  - Division durch null
  - Ungültige Indexwerte beim Zugriff auf Array-Elemente
- Solche Fehler, liegen in der Regel an einem Denkfehler des Programmierers
- Derartige Fehler sollte man nicht generell abfangen, da sonst Programmfehler nicht erkannt werden können
- Daher wurde die Unterklasse **`RuntimeException`** eingeführt, die Fehler beschreibt, die vom Programmierer behandelt werden können, aber nicht müssen
- Der Name **`RuntimeException`** ist jedoch seltsam gewählt, da **alle** Ausnahmen immer zur Laufzeit erzeugt, ausgelöst und behandelt werden

RuntimeException

- Oft vorkommende **`RuntimeExceptions`** sind:

Unterklasse von `RuntimeException`	**Was den Fehler auslöst**
`ArithmeticException`	Division durch null
`ArrayIndexOutOfBoundsException`	Indexgrenzen missachtet
`ClassCastException`	Typanpassung ist zur Laufzeit nicht möglich
`IllegalArgumentException`	Eine häufig verwendete Ausnahme, mit der Methoden falsche Argumente melden
`NullPointerException`	Falscher Zugriff auf eine Referenz mit dem Wert **`null`**

RuntimeExceptions



RuntimeException

- Eine **`RuntimeException`** muss nicht aufgefangen werden
- Da keine Behandlung erfolgen muss wird eine **`RuntimeException`** auch *unchecked exception* genannt
- Eine **`RuntimeException`** muss nicht in der **`throws`**-Klausel angegeben werden, es kann aber zu Dokumentationszwecken helfen

Error

- Fehler, die von der Klasse **`java.lang.Error`** abgeleitet sind, stellen Fehler dar, die mit der JVM in Verbindung stehen
- Anders dagegen die von **`java.lang.Exception`** abgeleiteten Klassen – sie stehen für eigene Programmfehler
- Beispiele für konkrete Error-Klassen sind **`AssertionError`**, **`AWTError`**, **`ThreadDeath`**, **`VirtualMachineError`**, **`InternalError`**, **`OutOfMemoryError`**, **`StackOverflowError`**, **`UnknownError`**
- Es ist möglich, ein **`Error`**-Objekt mit **`try/catch`** aufzufangen, da **`Error`**-Klassen Unterklassen von **`Throwable`** sind und sich daher genauso verhalten
- Das Auffangen ist in der Regel nicht sinnvoll, denn wenn die JVM einen Fehler anzeigt, bleibt offen, wie darauf sinnvoll zu reagieren ist
- Eigene Unterklassen von **`Error`** sollten keine Anwendung finden

Exceptions auslösen

- Soll eine Methode selbst eine Exception auslösen, muss zunächst ein **`Exception`**-Objekt erzeugt und dieses dann „geworfen“ werden
- Dazu dient das Schlüsselwort **`throw:`**

```
public void setRadius(double r) {
    if (r<0)
            throw new IllegalArgumentException();
    // weitere Implementation
}
```

- Beim Erzeugen einer Exception kann in der Regel auch eine Fehlermeldung angegeben werden:
  `throw new IllegalArgumentException(„Meldung“);`

Eigene Exceptions

- Eigene Exceptions sind immer direkte (oder indirekte) Unterklassen von **`Exception`**
- Es sollten zwei Konstruktoren implementiert werden:
  - *Default-Constructor*
  - *Custom-Constructor* mit formalem Parameter vom Typ **`String`**
- Würde man keine Unterklassen von **`Exception`** anlegen, könnte man einen Fehler später nicht mehr von anderen Fehlern unterscheiden
- Man sollte nicht zu inflationär mit den **`Exception`**-Hierarchien umgehen; in vielen Fällen reicht eine Standard-Ausnahme aus

Der Stack Trace

- Die JVM merkt sich auf einem Stapel (Stack Trace), welche Methode welche andere Methode aufgerufen hat
- Wenn die statische **`main()`**-Methode die Methode **`println()`** aufruft und die wiederum **`print()`**, sieht der Stapel zum Zeitpunkt von **`print()`** so aus:

```
print
println
main
```

- Ein Stack Trace ist im Fehlerfall nützlich, denn so lässt sich ablesen, wie die Aufruf-Geschichte war
- **`Throwable`** liefert Methoden zur Ausgabe des Stack Trace:
  - **`void printStackTrace()`** Schreibt das **`Throwable`** und anschließend den Stack-Inhalt in den Standardausgabestrom.
  - **`void printStackTrace(PrintStream s)`** Schreibt das **`Throwable`** und anschließend den Stack-Inhalt in den angegebenen **`PrintStream`**
  - **`void printStackTrace(PrintWriter s)`** Schreibt das **`Throwable`** und anschließend den Stack-Inhalt in den angegebenen **`PrintWriter`**

Der Stack Trace

- Doch auch ohne Ausnahme kann man sich den Stack Trace anschauen
- Der zum Zeitpunkt eines Methodenaufrufs aufgebaute Stack lässt sich mit einer **`Throwable`**-Instanzmethode erfragen
- Damit lassen sich Schlüsse über den Programmablauf rekonstruieren:

```
StackTraceElement[] trace =
                    new Throwable().getStackTrace();
```

- Dies erlaubt den Zugriff auf:
  - Den Dateinamen, in dem die Methode deklariert wurde
  - Die Programmzeile
  - Den Methodennamen
- Beispiel: **`stacktrace1`**

Zusammenfassung

- Assertions
- Exceptions
  - **`Error`**
  - **`RuntimeException`**
  - **`Exception`**
- Erzeugen und Werfen von Exceptions
- Implementierung eigener Exceptions
- Stack Trace

Ausblick

- Dateiinformationen
- Streams
- Filterstreams